Verificación de estructuras de datos enlazadas en Dafny

Verification of linked data structures in Dafny

Jorge Blázquez Saborido

Grado en Ingeniería Informática
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

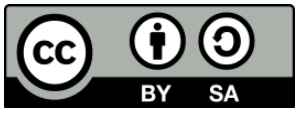
Trabajo de fin de grado

15 de junio de 2021

Directores:
Clara María Segura Díaz
Manuel Montenegro Montes

Que la vida iba en serio
uno lo empieza a comprender más tarde
–como todos los jóvenes, yo vine
a llevarme la vida por delante.

Jaime Gil de Biedma (1929–1990)



Contents

Resumen / Abstract v

1 Introduction **1**
1.1 Goals and results 2
1.2 Work plan 3

2 An overview of Dafny **5**
2.1 A tutorial on Dafny 5
2.2 Dafny types 10
2.3 ADTs in Dafny 15
2.4 Versions of Dafny 17

3 ADT verification methodology **19**
3.1 Calculated model 19
3.2 Structured or unstructured representation 20
3.2.1 `Insert` method with unstructured representation 22
3.2.2 Structured representation 25
3.3 Stratified representation 26

4 A layered approach to software verification **31**
4.1 Layer 1: ADTs 31
4.2 Layers 2 and 3: Implementations 35
4.3 Layer 4: Auxiliary classes 36

5 Examples **39**
5.1 Reordering 39
5.2 Finding summits 41

6 Iterators **47**
6.1 Verification of linked list iterators 47
6.2 Invalidation of linked list iterators 49

6.3 Using iterators for element insertion 52

7 Conclusions **57**
7.1 What I learned from this project 58
7.2 Difficulties 58
7.3 Future work 58

Resumen

El uso de la verificación formal está creciendo en las áreas de la industria del software donde se necesita la corrección total de un sistema. Dafny es un lenguaje de programación capaz de verificar programas que permite al programador especificar formalmente su código y que sea verificado por un demostrador de teoremas interno. Sin embargo, la verificación en Dafny no es una tarea sencilla. El programador tiene que dar una descripción detallada del comportamiento de su programa.

Hemos ideado una metodología para verificar *Tipos Abstractos de Datos* (TAD) implementados con estructuras de datos enlazadas almacenadas en la *heap* que permite reutilizar código y mantiene la abstracción absoluta de los detalles de implementación del TAD. Hemos desarrollado una biblioteca de TAD que sigue dicha metodología y hemos escrito ejemplos completos de programas que usan la biblioteca de manera cómoda. Hemos empezado la expansión de nuestra metodología para incluir iteradores, la cual ya ha dado resultados prometedores al aportar una especificación de iteradores que mantiene su validez cuando la modificación de la lista que están recorriendo no les afecta.

Palabras clave: verificación formal, estructuras de datos, tipos abstractos de datos, Dafny

Abstract

Formal verification is gaining adoption in the parts of the software industry where full correctness of a system is needed. Dafny is a programming language with verification capabilities that allows the programmer to formally specify their code and have it verified by an underlying theorem prover. Verification in Dafny, however, is not always an easy task. The programmer must give a detailed description of the behavior of the program.

We devise a novel methodology to formally verify *Abstract Data Types* (ADTs) implemented with heap-based linked data structures that allows code reuse and full abstraction from the implementation details of the ADT. We have developed a library of ADTs that follows that methodology and have written complete examples of programs using that library in an ergonomic way. We have started the expansion of the methodology to iterators, which has already given promising results by providing a specification of iterators that maintains their validity when the modification of the instance they are traversing does not affect them.

Keywords: program verification, data structures, abstract data types, Dafny

Agradecimientos

En primer lugar, quiero agradecer a mis directores de TFG el empeño que han puesto en este trabajo. En ningún momento me he sentido solo, siempre sabía que podía mandar un correo y que al poco tiempo tendríamos una reunión. Después de cada reunión los problemas se solucionaban o por lo menos eran más asequibles.

Quiero agradecer también a mis compañeros de carrera con los que he sufrido muchas prácticas, en especial a Dani, que siempre me consigue sacar una sonrisa; a Kike, cuya pasión por lo que le gusta me inspira; y a Jesús, que trabaja incansablemente para sacar lo que haga falta adelante. Y a todos, y a muchos que me dejo, por aguantar mi perfeccionismo cuando teníamos que repetir algo porque no me parecía suficiente.

Quiero agradecer a mi padre que me diese un manual de Small Basic, que todavía conservo, hace ya muchos años. Gracias a él no recuerdo un momento en mi vida en el que no hubiese informática alrededor mío. De igual forma, quiero agradecer a mi hermano Javi que me diese un manual de Python, con el que empecé mi aventura entre los lenguajes de programación. A mi madre la agradezco las conversaciones que tenemos en la cocina que me permiten seguir adelante y que echaré de menos.

Por último, quiero agradecer a mis amigos de los que no me separo. A Víctor porque cada vez que le veo su sonrisa hace que se me olviden mis problemas. A Edu por ser esa constante en mi vida con la que siempre puedo contar. A Pablo por esas conversaciones filosóficas que rozan el absurdo que me frustran y me dan la vida a la vez. Y a Georgia, porque no hay nadie en quien confíe más.

Chapter 1

Introduction

Formal verification is an established method to prove that the code a programmer writes matches the expected behaviour. It is driven by the need to build more robust systems. In contrast with other methods, like testing, it guarantees full correctness up to the correctness of the specification (if the specification reflects the expected behavior).

Formal verification has seen wide adoption in the hardware industry [20, 8] and it is also slowly gaining momentum in the software industry, with examples such as the seL4 microkernel [12] and the CompCert compiler [16], meant to be deployed on high-assurance systems. These examples show the demand for this level of correctness in software systems.

In order for a system to be formally verified, several approaches have been developed with varying degrees of automation. On one side of the spectrum, proof assistants help the proof engineer develop manual proofs of correctness of the system. Examples of proof assistants include Coq [4], Isabelle [11] and Agda [1]. On the other side of the spectrum, semiautomated theorem provers take the automatically generated *proof obligations* of the system and prove them. Examples of such provers are SMT solvers like Z3 [6], the underlying theorem prover that the language we will use during this work, Dafny [15], is based on. Note that even though proof assistants are essentially manual, they usually offer automation; and even if semiautomated theorem provers are mainly automatic, they take hints to help them verify all proof obligations in a reasonable time. Another approach to formal verification is model checking [3], where the states of a finite-state model of a program are checked to satisfy the specification. This process is fully automatic but it is not as scalable as other approaches.

1.1 Goals and results

This work is concerned with the verification of *Abstract Data Types* (ADTs). An ADT is a set of operations that manipulate elements of a given *abstract model*. This model can be realized in several ways, each of them leading to a specific concrete implementation. In this work we focus on mutable, heap-allocated ADTs, which are those that can usually be found in mainstream imperative languages. In this context, the specification of the operations becomes more complex, since such specifications have to be expressed in terms of the abstract model, while there is a *representation invariant* that links the model to the private details of the implementation. Moreover, some ADT operations modify the heap as a side effect, so their specification should also demarcate which parts of the heap might be affected. These challenges have been addressed in the context of Dafny [15, 14, 19], JML [9], Eiffel [18], Ada [7], among others. In the case of Dafny, previous approaches suffered from lack of encapsulation and modularity. Lack of encapsulation led to the loss of part of the abstraction level that is characteristic of ADTs, not letting different versions of the same ADT to be interchangeable. Lack of modularity meant that the same underlying data structure was not reused to implement different ADTs.

We set as a goal the development of a verification methodology of linked data structures in the form of ADTs that solves the problems found in the literature. This methodology should be the result of comparison of different approaches. These ADTs should be usable to code and verify real examples in an ergonomic way (not very different to what one would normally do in Dafny). As an extra goal, we would like to explore the verification of iterators in linked lists.

As a result, we have developed a library of heap-based linked data structures and the methodology is described in this document and in the accepted paper for the national event PROLE 2021 [2]. Our main contributions are (1) a layered approach to split our ADTs into different levels of abstraction, (2) alternative function-based definitions of the model and footprint of a data structure, which allows us to omit the corresponding ghost fields and their explicit update, (3) a stratified approach to the representation of an instance's footprint in order to support hierarchies of data structures, (4) a twofold interpretation of an ADT's footprint, as a generic unstructured set, and also as a implementation-dependent structured representation, and (5) an implementation of iterators with different invalidation policies.

1.2 Work plan

The phases of development of this work were:

- Learn Dafny: from July to September. The result of this phase can be found in Chapter 2.
- Apply previous approaches to verification of linked data structures: from August to September. The result of this phase can be found in Section 2.3.
- Explore alternative approaches: from October to December. The result of this phase can be found in Chapters 3 and 4.
- Implement different linked data structures following the methodology developed in the previous phase and give real world examples: from January to March. The result of this phase can be found in Chapters 4 and 5.
- Explore verification of list iterators: during April. The result of this phase can be found in Chapter 6.

Chapter 2

An overview of Dafny

Dafny [14] is an object-oriented programming language that focuses on program correctness by leveraging a program verifier under the hood. Programs are annotated with preconditions, postconditions, invariants and assertions that are sent to a program verifier based on an SMT solver [13] that makes sure that they are correct. These annotations are checked at compile-time to be correct for all inputs, unlike runtime checks in other imperative programming languages. This means that we can mathematically prove our programs are correct in all cases, not only on those we tested.

In this chapter we give a brief introduction on how to verify programs in Dafny and show all the features that are used throughout this work. Then we will continue to explain how ADTs are usually formalized in Dafny. To finish, we will discuss the different versions of Dafny that were used in the development of our library and what are their differences.

2.1 A tutorial on Dafny

Dafny is an OOP language and, like others, has classes as the main building block to write programs. Classes have fields and methods, as shown in Figure 2.1. The class `Counter` has a field `x` that can be increased with the `Inc` method and retrieved with `Get`. A new instance with the counter set to 0 can be created thanks to its constructor with the `new Counter()` expression. The `modifies` clause will be explained later.

This, however, can be done with any imperative object-oriented programming language. What sets Dafny apart is its ability to verify properties of our pro-

```
class Counter {
  var x: int;

  constructor()
  {
    x := 0;
  }

  method Inc()
    modifies this
  {
    x := x + 1;
  }

  method Get() returns (r: int)
  {
    r := x;
  }
}
```

Figure 2.1: Counter class

grams through the specification of preconditions and postconditions in method definitions, in the style of Hoare logic [10]. For example, we may want to ensure that the counter from the previous example never goes below zero. To do that we define a predicate on that class that will specify when our counter is valid, as shown in Figure 2.2. Then, if we set that predicate as the precondition (`requires` clause) and postcondition (`ensures` clause) of every method we can be sure that the counter never reaches an unwanted state.

Now we will continue with a more interesting example: a method `Fill` that sets the elements in a certain range of an array to a given value. In Figure 2.3 we write the implementation and specification of that method and Dafny will make sure that they match. To understand that method, however, we first need to know how a specification is written in Dafny.

Methods have three clauses: `requires` clauses (preconditions), `ensures` clauses (postconditions) and a `modifies` clause. The modifies clause specifies which memory locations might be modified during the execution of the method, in our case the array parameter. In the `Fill` method we have as precondition that the range $[l, r)$ we are given is well-formed, and as postconditions that we really set the elements in that range to the value `c` we receive while maintaining the rest of the elements the same. We write the postcondition with a `forall` expression, represented in this document with a $\forall$ symbol to improve readability. For an expression `e`, we write `old(e)` to refer to the value that expression had

```
class Counter {
  var x: int;

  predicate Valid()
    reads this
  { x ≥ 0 }

  constructor()
    ensures Valid()
  { x := 0; }

  method Inc()
    modifies this
    requires Valid()
    ensures Valid()
  { x := x + 1; }

  method Get() returns (r: int)
    requires Valid()
    ensures r ≥ 0
  { r := x; }
}
```

Figure 2.2: Counter class with validity

before the execution of the method.

The implementation of the method is very straightforward, the only difference to other languages is that we need to annotate the while-loop with invariants. In this case, that the positions in the range are set to c if we have already gone through them and that the rest have not changed. Dafny will make sure that these invariants are satisfied but it cannot generate them for us; we have to write them manually.

All the annotations that we added to that method (preconditions, postconditions, invariants) are compile-time only. That means that no checks will be made at run-time: the verifier has proved that they are always correct. Here we can clearly see the distinction between the implementation section of Dafny and the verification section. Now we will look into ways to write more complex specifications.

Sometimes the specification of a method cannot simply be expressed with the predefined functions of Dafny, but the language allows us to define new ones. For example, the Fibonacci sequence can be defined as a recursive function. To define it we have *functions*. Functions are part of the verification section: they are never executed (they are *ghost code*) and behave as mathematical functions. In Figure 2.4 we define the Fibonacci numbers and then use that definition to write the

```
method Fill(v: array<int>, l: int, r: int, c: int)
  modifies v
  requires 0 ≤ l ≤ r ≤ v.Length
  ensures ∀ k | 0 ≤ k < l • v[k] = old(v[k])
  ensures ∀ k | l ≤ k < r • v[k] = c
  ensures ∀ k | r ≤ k < v.Length • v[k] = old(v[k])
{
  var i := l;
  while i < r
    invariant i ≤ r
    invariant ∀ k | 0 ≤ k < l • v[k] = old(v[k])
    invariant ∀ k | l ≤ k < i • v[k] = c
    invariant ∀ k | r ≤ k < v.Length • v[k] = old(v[k])
  {
    v[i] := c;
    i := i + 1;
  }
}
```

Figure 2.3: Fill method

specification of a method that returns the nth Fibonacci number. Dafny will make sure that the imperative implementation matches the recursive specification.

Notice that we have added the `decreases` clause to the recursive function and to the while-loop. This clause is used to help Dafny prove termination, although it is usually omitted since Dafny can infer it in many cases.

Apart from functions and methods, we can also declare predicates, like the `Valid` predicate in Figure 2.2. These are simply syntactic sugar for functions that return a boolean.

One important difference between methods and functions is that methods are opaque: their implementation is not visible from outside. Functions, on the other hand, share their implementation to the outside world and we can prove properties about them that do not necessarily appear in their postconditions.

Sometimes we have a function that we would like to execute at runtime or, on the contrary, a method that we want to use in the specification of other methods. This is the task of `function methods`. These are defined exactly the same way we would define functions, but can appear both in the implementation and in the specification section (and will be executed if they appear in the former). Function methods are very useful, for example, in the conditions of if-statements and while-loops since Dafny does not allow method calls in that context. The `Fib` function in Figure 2.4 could be converted to a function method and be used as an alternative implementation of the Fibonacci numbers. During this work, however, we will limit the use of function methods to the bare

```
function Fib(n: nat): nat
  decreases n
{
  if n = 0 then
    0
  else if n = 1 then
    1
  else
    Fib(n-1) + Fib(n-2)
}

method Fibonacci(n: nat) returns (r: nat)
  ensures r = Fib(n)
{
  var i := 0;
  var x := 0;
  var y := 1;
  while i < n
    decreases n - i
    invariant i ≤ n
    invariant x = Fib(i)
    invariant y = Fib(i+1)
  {
    x, y := y, x + y;
    i := i + 1;
  }
  r := x;
}
```

Figure 2.4: Fibonacci sequence specification and implementation

minimum (methods that we want to use in conditions).

Another way to move a method from the implementation section to the verification section is by declaring it *ghost*. This, however, does not let us use it inside specifications. It is only a way to inform Dafny that we do not want the code of that method to be compiled; it is only present for verification purposes. This is useful, for example, when we want to prove a lemma that is necessary to verify other methods: once we call this method all its proofs will be added to the context. In fact, this is so common that writing `ghost method` and `lemma` is equivalent. In Figure 2.5 we show an example of a lemma present in our library: it proves that the `Rev` function (the function that reverses a sequence of elements) places the last element of the input (`xs[|xs|-1]`) as the first element of the output. Its proof uses the `calc` statement, which takes a list of expressions and asserts that each one is equal to the next. We can give helper annotations inside braces to help Dafny prove each step. In Section 2.2 we will go into more detail about sequences and other immutable datatypes.

Similarly to methods, fields in classes can be declared *ghost* and they will not be compiled.

Classses are not the only top level declaration possible in Dafny: traits are also available. Traits are equivalent to abstract classes in other languages. In Dafny, they let us abstract us away from implementation details. That is the reason traits are used pervasively in this work. Classes can extend a trait if they implement their interface, but they cannot extend other classes. Traits, on the contrary, can extend other traits [1]. In Figure 2.6 we show an example of a trait where both the `Cat` and `Bonobo` classes extend the `Animal` trait, each defining their respective number of legs.

2.2 Dafny types

Dafny types are divided between value types and reference types [5]. Value types are the basic primitive types (`int`, `nat`, `bool`, etc.) and the following collection immutable datatypes:

- Sets (`set<A>`): unordered collections of elements. The empty set is `{}`, a set with three elements is written `{1, 2, 3}` and we can express the set

[1] This feature is only available in Dafny 3. We will talk more about the different versions of Dafny in Section 2.4.

```
function Rev<A>(xs: seq<A>): seq<A>
  ensures |xs| = |Rev(xs)|
{
  if |xs| = 0 then
    []
  else
    Rev(xs[1..]) + [xs[0]]
}

lemma LastRev<A>(xs: seq<A>)
  requires |xs| > 0
  ensures Rev(xs) = [xs[|xs|-1]] + Rev(xs[..|xs|-1])
{
  if |xs| = 0 {
  } else if |xs| = 1 {
  } else {
    calc = {
      Rev(xs);
      = Rev([xs[0]] + xs[1..]);
      = Rev(xs[1..]) + [xs[0]];
      = { LastRev(xs[1..]); }
        [xs[|xs|-1]] + Rev(xs[1..][..|xs[1..]|-1]) + [xs[0]];
      = { assert xs[1..][..|xs[1..]|-1] = xs[1..|xs|-1]; }
        [xs[|xs|-1]] + Rev(xs[1..|xs|-1]) + [xs[0]];
      = [xs[|xs|-1]] + Rev([xs[0]] + xs[1..|xs|-1]);
      = [xs[|xs|-1]] + Rev(xs[..|xs|-1]);
    }
  }
}
```

Figure 2.5: Reverse function and a lemma about it

```
trait Animal {
  function NumberOfLegs(): nat
}

class Cat extends Animal {
  function NumberOfLegs(): nat
  {
    4
  }
}

class Bonobo extends Animal {
  function NumberOfLegs(): nat
  {
    2
  }
}
```

Figure 2.6: Animal taxonomy expressed as a trait and classes

union via the + operation: `{1, 2} + {3} == {1, 2, 3}`. We can say a set `r` is a subset of other set `s` by writing `r <= s`.

- Multisets (`multiset<A>`): similar to sets but they keep track of the multiplicity of each element. Multisets are built the same way as sets but they require the keyword `multiset`: `multiset{}`, `multiset{1}`, `multiset{1, 2}`, `multiset{1, 2, 3}`, etc.

- Sequences (`seq<A>`): an ordered collection of elements. The empty list is `[]` and a list of three elements can be written `[1, 2, 3]` that we will call `s`. We can extract a slice with the notation `s[i..j]`, for example, `s[1..2] == [2]`. We can omit the first or last index of the slice: `s[1..] == [2, 3]`, `s[..2] == [1, 2]`. We can also concatenate two lists with the + operator: `[1, 2] + [3] == [1, 2, 3]`. The multiset of the elements of `s` is `multiset(s)`.

- Other types that we do not use in this work like maps and infinite sets.

We can get the size of a collection `c` with `|c|`.

Reference types are different to value types in being allocated in the heap. When they are passed to a method they are passed by reference, which means that they can be modified (and, consequently, added to the `modifies` clause). Reference types include all the types of classes and the array type.

Arrays can be allocated with the `new T[n]` expression for a type `T` and length `n`. Their elements are accessed and modified as is usual in other programming languages and its length is obtained with `v.Length` for an array `v`. Like sequences, we can get the multiset of the elements of an array `v` with `multiset(v)`. We can convert an array to a sequence with `v[..]`. If we only want to get the sequence of a part of the array we can also write `v[1..4]`.

We have developed a small collection of functions that operate on these built-in datatypes to aid in the verification of methods in the rest of the library. It is located in the `Utils.dfy` file.

Every class type is a subtype of the type `object`, as shown in Figure 2.7. By default, variables of a class type are not nullable (i.e. `null` is not a valid value), but we can get the nullable type of a class by adding `?` to its name, as shown in Figure 2.8.

Dafny has a feature called *framing* that allows us to specify which parts of the heap a function (or a predicate) reads so that when that part of memory is not modified, the function's result remains the same (or the predicate remains

```
class A {
  constructor() {}
}

method main()
{
  var a: A := new A();
  var o: object := a;
}
```

Figure 2.7: Every class is a subtype of `object`

```
class A {}

method main()
{
  var a: A? := null;
}
```

Figure 2.8: Nullable classes

valid). This feature is controlled with the `modifies` clause introduced earlier, only available for methods; and the `reads` clause, only available for functions and function methods. To illustrate this, in Figure 2.9 we define a predicate that specifies if an array is sorted. Then, after we call a method that modifies only one of the two sorted arrays, Dafny can prove that the unmodified array stays sorted but not the other. Notice that, in reality, none of the arrays are being modified since the method body is empty, but its specification says that it could be modified and Dafny will treat is as such since methods are opaque.

We can have fine-grained control over which memory locations a function reads with the backtick notation. For example, a function with the clause `reads x`data` will only read the `data` field of the object `x`. This will be useful in exceptional circumstances, but we usually include full objects in the `reads` clause.

Lastly, we can specify which objects will be newly allocated during the execution of a method through the use of the `fresh` keyword. In Figure 2.10 we specify a method that returns a new array (a *fresh* array) that is sorted. This method can be implemented with a simple while-loop.

```
predicate Sorted(v: array<int>)
  reads v
{
  ∀ i | 0 ≤ i < v.Length-1 • v[i] ≤ v[i+1]
}

method DoNothing(v: array<int>)
  modifies v
{}

method Main()
{
  var v := new int[3];
  v[0] := 0; v[1] := 1; v[2] := 2;
  assert Sorted(v);

  var w := new int[3];
  w[0] := 0; w[1] := 1; w[2] := 2;
  assert Sorted(w);

  DoNothing(w);
  assert Sorted(v);
  // assert Sorted(w);  // Dafny cannot prove this assertion
}
```

Figure 2.9: Framing example

```
method NewSorted(n: nat) returns (v: array<int>)
  ensures fresh(v)
  ensures Sorted(v)
  ensures v.Length = n
```

Figure 2.10: Specification of a method that returns a fresh sorted array

2.3 ADTs in Dafny

Now that we know the general concepts of how to verify programs in Dafny, we will continue by explaining how ADTs are usually defined in Dafny. Later, in Chapter 3, we will explore different ways to implement and define ADTs, our main contribution; but first we will review how they have been defined in the literature [15]. The verification of ADTs in Dafny is based on three definitions:

- Representation (footprint): the set of objects that each instance owns and uses to implement the interface. Mutators modify this representation. It is usually expressed as a set and stored in a ghost field:

```
ghost var repr: set<object>
```

- Model: the formal interpretation we give to the ADT. It should be a functional datatype because it is used for verification purposes. The model of a linear ADT is a sequence, provided by Dafny's immutable datatype `seq`:

```
ghost var model: seq<A>
```

- Representation invariant: it is the property that delimits which instances denote a value of the model. In Dafny it is represented with a predicate that depends on the representation:

```
predicate Valid()
  reads this, repr
```

In Figure 2.11 we show an example implementation of a singly linked list that has those three definitions. The representation holds the current node and all the nodes that come after it. The model is the `data` field of the current node and of the nodes after it. The representation invariant ensures that there are no loops in the links between nodes by forcing the representation to decrease from one node to the next. It also makes sure that the model is the sequence of values contained within the nodes.

In the constructor of the `Node` class we specify that all the newly added elements to the representation are fresh with a `forall` expression. We could use set operations to express that the set of newly added elements to the representation is fresh, but Dafny verifies proof obligations more easily with the more verbose `forall` expression, so we will follow that convention from now on.

As we said, our methodology differs significantly from this one, but this code is also present in the source code of our library [2] for comparison purposes and as the foundation of the rest of the data structures.

[2] In the `aux/LeinoList.dfy` file

```
class Node<A> {
  ghost var repr: set<Node<A>>;
  ghost var model: seq<A>
  var data: A;
  var next: Node?<A>;

  constructor(x: A, next: Node?<A>)
    requires next ≠ null ⟹ next.Valid()
    ensures Valid()
    ensures model = [x] + (if next = null then [] else next.model)
    ensures ∀ x | x in repr - (if next = null then {} else next.repr) • fresh(x)
  {
    this.data := x;
    this.next := next;
    if next = null {
      this.repr := {this};
      this.model := [data];
    } else {
      this.repr := {this} + next.repr;
      this.model := [data] + next.model;
    }
  }

  predicate Valid()
    reads this, repr
  {
    ∧ this in repr
    ∧ (if next = null then
         repr = {this} ∧ model = [data]
       else
         ∧ next in repr
         ∧ repr = {this} + next.repr
         ∧ model = [data] + next.model
         ∧ this ∉ next.repr
         ∧ next.Valid()
       )
  }
}
```

Figure 2.11: Node class

2.4 Versions of Dafny

Development of the library in this work began with Dafny 2.7. In that version, traits were not well-polished yet and we arrived at some limitations with them, like not being able to define subtraits or not having polymorphism. During the course of our work, however, Dafny version 3.0 was released, with the addition of subtraits and trait polymorphism, fixing two sticking points with them. However, there is still one missing feature that we need in order for us to finish our work on iterators: downcasting. In Chapter 6 we go deeper into why. This feature, however, is work in progress.

Dafny Version 3.1 was released shortly after Dafny 3.0 but we have not used it since it was very unstable. For example, it did not connect well to our IDE.

The code of our library can be verified with Dafny 3.0, except for the `CircularDoublyLinkedList` that we have not ported yet from Dafny 2.7 and only verifies in that version.

Chapter 3

ADT verification methodology

In Section 2.3 we described the three definitions needed to verify ADTs in Dafny and showed how they have been fulfilled in the literature. In this chapter we will describe how we have implemented those definitions and what are the advantages of our methodology with respect to other approaches.

During this chapter we will add incremental changes to the example in Figure 2.11 until we reach the final solution. Many of the intermediate steps are present in our library, although here they have been modified for illustration purposes. In the library we have defined several data structures [1]: singly linked lists, doubly linked lists, their respective versions with a pointer to the last node and circular doubly linked lists. All those data structures share the same verification principles but applied to their pecularities. We will focus on the singly linked list example because it is simple but includes all the problems that justify our decisions.

The changes described in this chapter make the verification of complex ADTs possible with less proof work. They also allow code reuse, which aids in software mantainability.

3.1 Calculated model

The first change we are going to add to our implementation of singly linked lists is regarding the model. In Figure 2.11 we can see that we have to manually update the model in the constructor so that it matches the state of the ADT. That process has to be done in each method that modifies the data structure and can become

[1] They can be found in the `src/linear/aux/` folder.

cumbersome. It is not a roadblock to our verification goals but, as we will see, it is not really necessary.

The alternative to having a ghost field that stores the model is defining a function `Model()` that computes it on demand, as shown in Figure 3.1. This function is in the verification section so we will not have any runtime performance penalty. Now the style of verification is different: instead of modifying the model, proving that it is modified correctly (i.e. it satisfies the representation invariant of the ADT) and proving that the new model is what we expect; we will prove directly that the sequence that results from calling the `Model()` function is what we expect after the modification. We have freed us from proving that the model is correct, **the model is always correct now** since it is not present in the definition of the representation invariant. In fact, the computation of the model depends on the fact that the list is `Valid`: in order for the recursion of `Model` to be well-formed, the representation must decrease at each recursion step (exactly what the representation invariant says).

A calculated model has proved useful and we would like to apply the same principle to the representation, but a calculated representation does not seem possible. Remember that the representation of each node is exactly the set of nodes that come after it and itself. If we were to define a function `Repr()` that returns the representation of a given node, we would need to explore the next nodes. However, every function in Dafny must be well-formed, i.e. some parameter must decrease for every recursive call. In the definition of `Model()` the representation decreases, but the `Repr()` does not have any field that decreases after each recursive step. We could implement `Valid()` and `Repr()` mutually recursively, but then the decrease metric would be `Repr()` itself. In Figure 3.2 we try doing that but the termination checker rejects the program. We have not tried other methods to define a calculated representation since the modifications in the next section have worked well, but we will keep defining the representation as a function. This function will not be recursive, it will just take the local field (`repr` until now) and return it.

3.2 Structured or unstructured representation

Until now we have defined the representation as a set of objects. This is adequately abstract for the interface of the ADT but sometimes we need more information about our data structure. For example, in the case of a linear collection, like a linked list, a set does not inform us of the order of the nodes. This is not a

```
class Node<A> {
  ghost var repr: set<object>;
  var data: A;
  var next: Node?<A>;

  predicate Valid()
    reads this, repr
  {
    ∧ this in repr
    ∧ (if next = null then
         repr = {this}
       else
         ∧ next in repr
         ∧ repr = {this} + next.repr
         ∧ this ∉ next.repr
         ∧ next.Valid()
       )
  }

  function Model(): seq<A>
    decreases repr
    reads this, repr
    requires Valid()
  {
    if next = null then
      [data]
    else
      [data] + next.Model()
  }

  constructor(d: A)
    ensures Valid()
    ensures ∀ x | x in repr - {this} • fresh(x)
    ensures Model() = [d]
  {
    repr := {this};
    data := d;
    next := null;
  }
}
```

Figure 3.1: `Node` class with calculated model. This code is available in the `SinglyLinkedListWithRecursion.dfy` file in our library.

```
predicate Valid()
  decreases Repr()
  reads this, Repr()
{
  ∧ this in Repr()
  ∧ (if next = null then
       Repr() = {this}
     else
       ∧ next in Repr()
       ∧ Repr() = {this} + next.Repr()
       ∧ this ∉ next.Repr()
       ∧ next.Valid()
     )
}

function Repr(): set<object>
  decreases Repr()
  reads this, Repr()
  requires Valid()
{
  if next = null then
    {this}
  else
    {this} + next.Repr()
}
```

Figure 3.2: An attempt at defining a calculated representation

problem for simple methods, but when we want to expand our data structure to verify complex mutations on our data structure the proof work needed begins to be too big and unmantainable. During this section we are going to explain the implementation of one such method, the `Insert` method, in the class shown in the last section. Then, we will explain a different approach to defining our representation that will greatly simplify our proof work on that same method.

3.2.1 `Insert` method with unstructured representation

In Figure 3.3 we show the interface of the `Insert` method. This method is only meant to be used internally; that is why it receives a `Node`. If we wanted to expose it we would need iterators, as shown in Chapter 6. This method is defined in the `List` class, a class with a field `head` of type `Node?<A>`. Its specification uses an auxiliary function `Seq.Insert` that, given an element `x`, a sequence `xs` and an index `i`, returns `xs[..i] + [x] + xs[i..]`.

Let us first show the source code of the implementation without any verification annotations in Figure 3.4. It is a very simple method in terms of imple-

```
method Insert(mid: Node<A>, x: A)
  modifies this, mid
  requires Valid()
  requires mid in Repr()
  ensures Valid()
  ensures Model() = Seq.Insert(x, old(Model()), old(GetIndex(mid))+1)
  ensures ∀ x | x in Repr() - old(Repr()) • fresh(x)
```

Figure 3.3: Specification of `Insert` method

```
var n := new Node(x);
n.next := mid.next;
mid.next := n;
```

Figure 3.4: Body of `Insert` method without verification annotations

mentation, but we have to prove several things to verify it. First, we must set the representation of the new node created to be the set of all the nodes that come after it. After changing the `mid` node we have invalidated all the nodes that come before it: their representations do not include the newly added node. The bulk of the verification work is focused on modifying each node's representation from the head until `mid` to add the new node. All these modifications are only necessary in the verification section of Dafny, they will not be compiled and are defined as ghost code. In Figure 3.5 we show the `Repair` method that receives a node and all the nodes that come before it. It then modifies (repairs) the previous nodes to make them valid again. In its specification we use the `ReprAux` function that returns the empty set if it receives `null` or simply returns the representation of the node it is given. We also use the backtick notation `n`repr` to express that only the `repr` field of each node will me modified.

Now that we know how to repair the previous nodes the only thing left to do is group them in a sequence and wire the proofs together. In Figure 3.6 we show part of the interface of the `TakeSeq` function that returns the list of nodes between two nodes and in Figure 3.7 we show the final implementation of the `Insert` method, marked with comments in the lines that are ghost code. Some postconditions from Figure 3.3 have not been verified, notably the last two, since we stopped development of this class. Instead, we tried the alternative shown in the next section and found it was better to work with.

```
ghost method Repair(prevs: seq<Node<A>>, mid: Node?<A>)
  modifies set n | n in multiset(prevs) • n`repr
  requires mid ∉ multiset(prevs)
  requires ∀ n | n in prevs • n ∉ ReprAux(mid)
  requires ∀ i, j | 0 ≤ i < j < |prevs| • prevs[i] ≠ prevs[j]
  requires ∀ i | 0 ≤ i < |prevs|-1 • prevs[i].next = prevs[i+1]
  requires prevs ≠ [] ⟹ prevs[|prevs|-1].next = mid
  requires ValidAux(mid)
  ensures ValidAux(mid)
  ensures ∀ n | n in multiset(prevs) • n.Valid()
{
  if prevs ≠ [] {
    var prev := prevs[|prevs|-1];
    assert prev in multiset(prevs);
    prev.repr := {prev} + ReprAux(mid);
    assert prevs = prevs[..|prevs|-1] + prevs[|prevs|-1..|prevs|];
    assert multiset(prevs[..|prevs|-1]) ≤ multiset(prevs);
    Repair(prevs[..|prevs|-1], prev);
  }
}
```

Figure 3.5: Specification and implementation of the `Repair` ghost method

```
static function TakeSeq(head: Node<A>, mid: Node<A>): (res: seq<Node<A>>)
  reads ReprAux(head)
  reads ReprAux(mid)
  requires ValidAux(head)
  requires ValidAux(mid)
  requires mid in ReprAux(head)
  ensures res ≠ [] ⟹ res[0] = head ∧ res[|res|-1].next = mid
  ensures ∀ i | 0 ≤ i < |res|-1 • res[i].next = res[i+1]
  ensures ∀ i, j | 0 ≤ i < j < |res| • res[i] ≠ res[j]
```

Figure 3.6: Specification of the `TakeSeq` function

```
method Insert(mid: Node<A>, x: A)
  modifies Repr()
  requires Valid()
  requires mid.Valid()
  requires mid in Repr()
  ensures Valid()
  ensures mid.Valid()
  ensures fresh(mid.next)
{
  var n := new Node(x);
  ghost var prevs := TakeSeq(head, mid);
  assert ∀ i | 0 ≤ i < |prevs|-1 • prevs[i].next = prevs[i+1];
  assert prevs ≠ [] ⟹ prevs[|prevs|-1].next = mid;
  n.next := mid.next;
  /*GHOST*/ n.repr := ReprAux(mid.next) + {n};
  mid.next := n;
  /*GHOST*/ mid.repr := {n} + mid.repr;
  /*GHOST*/ Repair(prevs, mid);
}
```

Figure 3.7: Specification and implementation of the `Insert` method

3.2.2 Structured representation

In this section we are going to explore a different way to express the representation of our data structure. This time the field we use to store the representation will be of type `seq<Node<A>>`. We call this *structured representation*, because the representation is more than just objects in memory, it is an immutable data structure that imposes the relative order between nodes. In Figure 3.8 we show the code for the `Node` class. This class does no longer have a representation, a representation invariant or a model, it is just a value object. In Figure 3.9, however, we show part of the source code of class `List` that represents the singly linked list. In that class we do have all the elements we expect from an ADT, with the difference that the representation is expressed as a sequence under the hood and then converted into a set in the `Repr` function.

Before we explore the new verification of the `Insert` method we need some general lemmas about our data structure. In Figure 3.10 we show the specification of two lemmas: the `DistinctSpine` lemma proves that each node in the spine is unique; this can be proved just from the definition of the representation invariant; and the `ModelRelationWithSpine` establishes the relation between the model and the spine so that, once we prove theorems about our spine, proving theorems about our model is trivial.

Now we can study the definition of the `Insert` method in Figure 3.11. The implementation code is the same and we just need some annotations before and

```
class Node<A> {
  var data: A;
  var next: Node?<A>;

  constructor(data: A, next: Node?<A>)
    ensures this.data = data
    ensures this.next = next
  {
    this.data := data;
    this.next := next;
  }

  predicate IsPrevOf(n: Node<A>)
    reads this
  {
    next = n
  }
}
```

Figure 3.8: Definition of the `Node` class.

after it. In terms of code size, it is very similar to what he had with the unstructured representation, but there is a crucial difference: the `DistinctSpine` and the `ModelRelationWithSpine` lemmas are not specific lemmas about this method (unlike the `Repair` method with the unstructured representation). On the contrary, they are just boilerplate code that we add to almost all methods so that Dafny can prove the postconditions with the help of some annotations.

The structured representation allows us to leverage the built-in axioms regarding sequences in Dafny, instead of having to manually prove every method from scratch. We think that this methodology can be applied to data structures of different shapes, for example, trees. Trees are not a built-in datatype of Dafny, so we would need to define an algebraic datatype for them. Some automation will be lost, but we think the ability to reason in terms of abstract trees instead of pointers between nodes will make verifying those data structures easier.

The code from this section is available in the `SinglyLinkedListWithSpine.dfy` file but both `DoublyLinkedList` and `CircularDoubleLinkedList` also use this kind of representation.

3.3 Stratified representation

Up to this point we have defined data structures that are useful for the implementation of different ADTs, so we would like to reuse them. But, when we try

```
class List<A> {
  var head: Node?<A>;
  ghost var spine: seq<Node<A>>;

  function Repr(): set<object>
    reads this
  {
    set x | x in spine
  }

  predicate Valid()
    reads this, Repr()
  {
    ∧ (∀ i | 0 ≤ i < |spine|-1 • spine[i].IsPrevOf(spine[i+1]))
    ∧ (if head = null then
        spine = []
      else
        spine ≠ [] ∧ spine[0] = head ∧ spine[|spine|-1].next = null
      )
  }

  static function ModelAux(xs: seq<Node<A>>): seq<A>
    reads set x | x in xs • x`data
  {
    if xs = [] then
      []
    else
      assert xs[0] in xs;
      assert ∀ x | x in xs[1..] • x in xs;
      [xs[0].data] + ModelAux(xs[1..])
  }

  function Model(): seq<A>
    reads this, spine
    requires Valid()
  {
    ModelAux(spine)
  }
}
```

Figure 3.9: Definition of `List` class.

```
lemma DistinctSpine()
  requires Valid()
  ensures ∀ i, j | 0 ≤ i < j < |spine| • spine[i] ≠ spine[j]
lemma ModelRelationWithSpine()
  requires Valid()
  ensures |spine| = |Model()|
  ensures ∀ i | 0 ≤ i < |spine| • spine[i].data = Model()[i]
```

Figure 3.10: `DistinctSpine` lemma and `ModelRelationWithSpine` lemma

```
{ // GHOST
  DistinctSpine();
  ModelRelationWithSpine();
}
var n := new Node(x, mid.next);
mid.next := n;
{ // GHOST
  ghost var i :| 0 ≤ i < |spine| ∧ spine[i] = mid;
  spine := spine[..i+1] + [n] + spine[i+1..];
  ModelRelationWithSpine();
}
```

Figure 3.11: Implementation of the `Insert` method with structured representation

```
include "SinglyLinkedListWithSpine.dfy"

class Stack {
  var list: List<int>;

  function Repr(): set<object>
    reads this, list
  {
    list.Repr()
  }
}
```

Figure 3.12: A class, `Stack`, that uses the singly linked list internally

to build an ADT based on top of another ADT, we need to include the representation of the child ADT into the representation of the parent ADT. In order for us to do that, we need to add to the `reads` clause the field where we have the child ADT stored, as shown in Figure 3.12. This is fine, but what if we are building a general interface for stacks? We cannot know in advance which fields the `Repr` function will read from since those fields are implementation dependent. We could express the representation as a field, but then the structured representation of the previous section would not be possible.

In Figure 3.13 we introduce the concept of *stratified representation*. Instead of setting an upper limit to the number of levels of abstraction that are allowed, we build a family of representations `ReprFamily` that are built on top of the other: each level can read from the objects of the previous levels and each level is included in all the next levels. We can expect to have in the first level the fields of the parent ADT, in the second the representations of those fields (if they are ADTs), in the third level objects that depend on the representation of the child

```
function ReprDepth(): nat
  ensures ReprDepth() > 0

function ReprFamily(n: nat): set<object>
  decreases n
  requires n ≤ ReprDepth()
  ensures n > 0 ⟹ ReprFamily(n) ≥ ReprFamily(n-1)
  reads this, if n = 0 then {} else ReprFamily(n-1)

function Repr(): set<object>
  reads this, ReprFamily(ReprDepth()-1)
{
  ReprFamily(ReprDepth())
}
```

Figure 3.13: Stratified representation interface

ADTs, and so on. An example class that would use the singly linked list internally is shown in Figure 3.14. In level 0 it has the list that it uses to implement the stack and level 1 adds the representation of that list. The `ReprFamily` groups all the levels and, finally, the `Repr` function takes the biggest set of that family.

Using the stratified representation we can specify the interfaces of ADTs as traits and not worry about not being general enough. The implementations of these ADTs will be able to reuse as many auxiliary classes as they need. In Chapter 4 we list all the ADTs we have defined using this methodology.

```
class Stack {
  var list: List<T>;

  function ReprDepth(): nat
  {
    1
  }

  function Repr0(): set<object>
    reads this
  {
    {list}
  }

  function Repr1(): set<object>
    reads this, Repr0()
  {
    {list} + list.Repr()
  }

  function ReprFamily(n: nat): set<object>
    decreases n
    ensures n > 0 ⟹ ReprFamily(n) ≥ ReprFamily(n-1)
    reads this, if n = 0 then {} else ReprFamily(n-1)
  {
    if n = 0 then
      Repr0()
    else if n = 1 then
      Repr1()
    else
      ReprFamily(n-1)
  }

  function Repr(): set<object>
    reads this, ReprFamily(ReprDepth()-1)
  {
    ReprFamily(ReprDepth())
  }
}
```

Figure 3.14: An example of a stratified representation

Chapter 4

A layered approach to software verification

Up until now we have mainly focused on verifying ADTs, but not on how we expose their methods to users that may want to write algorithms with them or extend their functionality. In this chapter we describe the general structure of this work, composed of several layers that go down from the implementation level up to the user-facing interfaces, with some layers in between, as depicted by Figure 4.1.

Since a linked list can be used to implement different ADTs, we include all the variations of linked lists into layer 4 to be reused by the layers on top. In layer 3 we use those auxiliary classes to implement the ADT functionality, and we use layer 2 as the user-facing interface of those implementations, in the form of traits. In layer 1 we define the ADTs omitting any implementation details, contrary to what is done in layer 2. We will go deeper into the differences between layers 1 and 2 in Section 4.2 and exemplify them in Chapter 6.

4.1 Layer 1: ADTs

Layer 1 is composed of traits that constitute the interfaces of our ADTs. This corresponds to the interface layer in standard software libraries. Here, however, the interface does not only include type signatures. The complete formal semantics of the ADT is included as specification annotations in the interface. This has a double purpose: (a) ensure that the implementations of the interface conform to the specification and (b) let the users of the library prove properties of their

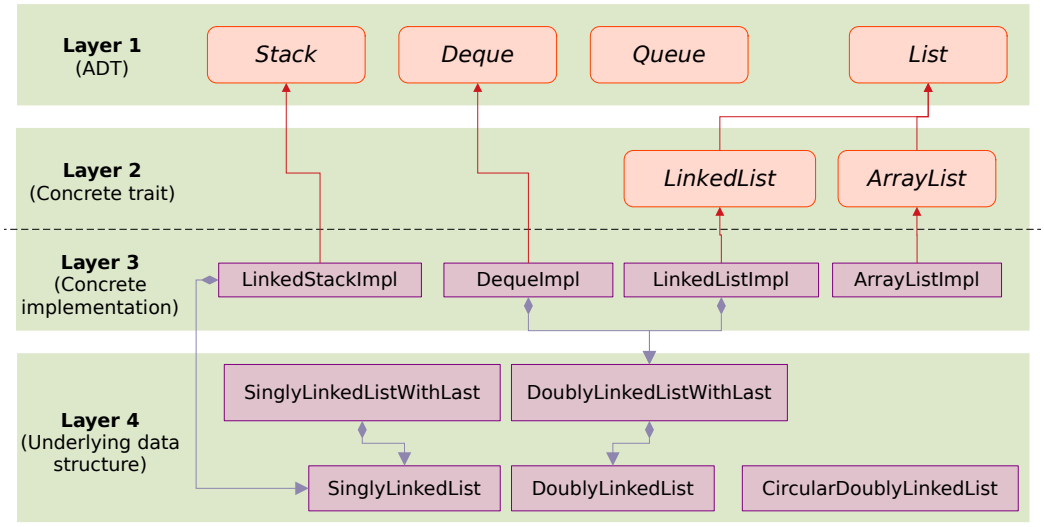


Figure 4.1: Abstraction layers for ADT verification

code. The ADTs that we have formalized in this layer are the stack, the queue, the deque and the list [1].

As we will see in Section 4.3, many implementations of the same ADT have been added to the library. Having such a strong interface lets us be completely sure that they represent the same ADT because they satisfy the same formal semantics.

In Chapter 5 we will see some examples using this library. These examples use the interfaces of this layer since using interfaces in the layers below would make their code less general. However, using a very general interface is not an impediment to specifying whole problems and prove their implementation is correct.

The code in layer 1 uses a set of conventions pervasively. We have developed these conventions to ensure that our ADTs are usable and extensible. These conventions are very repetitive: we need to add them to every precondition and postcondition in order for them to achieve their goal. This makes the exposition of algorithms difficult, so they are largely omitted in the code excerpts of this document, but they are present in almost all of the methods in the source code.

The code that we have to add to every method is usually regarded as *boiler-*

[1] They can be found in the `src/linear/adt/` folder.

plate and there is a feature in Dafny that is meant to reduce it: `autocontracts` [14]. We do not use that feature since it assumes following a different methodology than ours. For example, it automatically adds a ghost field `Repr` of type `set<object>`, but in previous chapters we have showed how different approaches to expressing the representation are better for verification automation. In order for us to use `autocontracts`, this feature would need to be changed or extended to accept our methodology.

Our main concern with ADT interfaces is encapsulation. We do not want any implementation details to be exposed. If they were, verification would probably be easier for the user [2], but their code would not be portable to different implementations of the same ADT. Wanting complete encapsulation, however, has led us to use traits, a recently added feature in Dafny that has some unexpected behaviors. We will now focus on adding the appropriate preconditions and postconditions to solve that.

We will begin by dividing the specification section of each method into four parts, as shown in the `Pop` method of the stacks in Figure 4.2:

1. The framing `modifies` clause: if the method is a mutator we must give the representation as the set of objects that will be modified. Sometimes we modify other objects too, like input parameters.

2. The preconditions and postconditions concerning the model of the ADT. This is specific to each method and it is what describes its semantics.

3. The boilerplate preconditions and postconditions concerning the invariant of the representation: all the ADT methods require the validity of the object as a precondition and mutators must ensure that it is valid after the mutations with the corresponding postcondition.

4. The boilerplate preconditions and postconditions concerning memory allocation.

Items 1 through 3 are simple, but we should dive deeper into preconditions and postconditions about memory allocation. First, in the `Pop` method of Figure 4.2, we make sure that every new object added to the representation is *fresh* (i.e. allocated during the execution of the method). This does not necessarily imply that the representation grows or shrinks after the execution of the method. It just

[2] This is because when Dafny knows implementation details, the verifier has more information and it can solve its goals with greater ease.

```
method Pop() returns (x: int)
  // 1. Framing clause
  modifies Repr()

  // 2. ADT preconditions and postconditions concerning the model
  requires ¬Empty()
  ensures [x] + Model() = old(Model())

  // 3. Representation invariant
  requires Valid()
  ensures Valid()

  // 4. Memory allocation
  ensures ∀ x | x in Repr() - old(Repr()) • fresh(x)
  ensures ∀ x | x in Repr() • allocated(x)
```

Figure 4.2: `Pop` method of the stack

states that, if it grows, it only does with newly allocated objects. For example, a stack implemented with an array will not decrease its representation, it will stay the same. A stack implementation using a linked list, however, will decrease its representation by freeing the node that it popped.

The second postcondition about memory allocation is more subtle, since it seems obvious. In Dafny, like in many languages with automatic memory management, every pointer we have access to points to a valid memory location (i.e. for every object `x`, `allocated(x)` can be proved). That means that there is no such thing as dangling pointers, all of them point to allocated objects. If that is the case, why do we need to specify that the method must make sure that every object is allocated? We add that proposition as a postcondition because Dafny will not try to prove it even when it needs it to verify other proof obligations (it is not part of the built-in axioms of Dafny). That proposition is needed to verify, for example, that one instance of an ADT has remained valid after a different instance has been modified. These simple uses of ADTs must be supported by our work since they appear in the examples in Chapter 5, so we add it as a postcondition to every method (and sometimes as a precondition too) to support them.

Verifying that proposition is trivial: we can prove it with the lemma in Figure 4.3. We could call the `Obvious` lemma after every method call, but we prefer the postcondition approach since it does not clutter the user's code, only some lines are needed at the end of the specification of methods.

It is also worth noting that we only have problems with allocatedness because we express the representation as a function that returns a set. If it was simply a

```
lemma Obvious(s: set<object>)
  ensures ∀ x | x in s • allocated(x);
{}
```

Figure 4.3: The `Obvious` lemma

ghost field of type `set<object>` we would not need to add that postcondition since Dafny is able to automatically prove the allocatedness of all the objects in that set. However, as has been noted, we want to abstract ourselves from different implementation strategies of the representation.

A more in-depth discussion about this topic was held in Dafny's issue tracker in an issue we opened for that purpose [3].

4.2 Layers 2 and 3: Implementations

Layers 2 and 3 correspond to the implementation layer in standard libraries. Dafny does not have the feature of hiding private and public definitions, so we use traits to enforce encapsulation. These two layers would be merged in other languages but in Dafny we need layer 2 to represent the interface of the class and layer 3 to be its implementation. The implementations included in these layers are [4]:

- For the stack ADT: the `LeinoStack` (implemented with the methodology that Rustan Leino [15] uses, as explained in Chapter 2), the `LinkedStack` (implemented using a singly linked list) and the `ArrayStack` (implemented with an array).
- For the queue ADT: `SinglyLinkedQueue` and `DoublyLinkedQueue` (implemented with a singly and doubly linked list, respectively), and the `ArrayQueue` (with an array).
- For the deque ADT we have two implementations, one simply called `Deque` using a doubly linked list, and one that uses an array called `ArrayDeque`.
- For the list ADT: the `LinkedList` (using a doubly linked list) and the `ArrayList` (using an array).

[3] Issue 1552 in Dafny's GitHub repository `https://github.com/dafny-lang/dafny/issues/1152`

[4] They can be found in the `src/linear/impl/` folder.

The traits that constitute layer 2 are subtraits of the traits in layer 1. In fact, they are usually the same trait [5] since there are no notable implementation details that want to be exposed. In the case of lists, however, the behaviour of their iterators depends on the implementation. Iterators in layer 1 are very loosely specified, we can only prove basic properties of them using that layer. In layer 2, on the other hand, we can take advantage of knowing how the ADT is implemented under the hood to prove complex properties of iterators. This will be further explained in Chapter 6.

Traits in layer 2 must be subtraits of traits in layer 1. That means that the postconditions of a method in layer 1 must be weaker than the postconditions of that method in layer 2, and the preconditions of a method in layer 1 must be stronger than the preconditions of that method in layer 2 (i.e. behavioral subtyping [17] is preserved). This is checked automatically by Dafny.

Regarding the relation between layers 2 and 3, there is a one-to-one correspondence between the traits in layer 2 and the classes in layer 3. Each trait in layer 2 is extended by a single class in layer 3. Moreover, both contain the same methods and the same specifications. The only difference is that layer 3 contains implementations and concrete definitions of invariants, whereas layer 2 does not. A library user who wants to rely on implementation-dependent specifications should use the traits of layer 2 instead of those of layer 1, but layer 3 is not meant to be directly accessed by the user, except when creating objects with `new`, and even such tasks could be delegated to factory methods. The distinction between layers 2 and 3 is motivated by the need of abstracting the user from the representation invariants explained in Chapter 3. The specifications in layer 2 do not expose the representation invariant to the outside world. They only assert that those invariants are preserved, but nothing else. However, classes in layer 3 do contain the definition of the representation invariant as predicates, which are not opaque, as explained in Chapter 2. If a library user mentions those classes explicitly in the types of their variables, the prover might try to use the internal invariants to verify the program, thus breaking encapsulation.

4.3 Layer 4: Auxiliary classes

Finally, layer 4 implements common underlying data structures that form the basis of the implemented ADTs, so that most of the methods in layer 3 just delegate

[5] In that case, layer 2 is omitted and the class in layer 3 extends the trait in layer 1 directly.

their implementations to those in layer 4. It uses the verification techniques explained in Chapter 3. This layer exists to facilitate code reuse, since there could be several ADTs relying on the same data structure. This is the case of the list and deque ADTs, which can be implemented by means of circular doubly linked lists. There is also code reuse among the data structures in this layer. For example, the implementation of `SinglyLinkedListWithLast` consists of an instance of `SinglyLinkedList` and a pointer to the last node of the linked list. In total, these are the classes that have been implemented [6]:

- `LeinoList`: the singly linked list as implemented by Rustan Leino [15].
- Singly linked lists: beginning with an implementation with structured representation (`SinglyLinkedListWithSpine`) and unstructured representation (`SinglyLinkedListWithRecursion`), we continue implementing the `SinglyLinkedListWithLast` by delegating to the `SinglyLinkedListWithSpine` class.
- Doubly linked list: this time we define the `DoublyLinkedList` with structured representation from the start and use it to implement the `DoublyLinkedListWithLast` class. We also implement a `CircularDoublyLinkedList`.

[6] They can be found in the `src/linear/aux/` folder.

Chapter 5

Examples

In this chapter we will explore the use of the ADTs described in previous chapters. Verified data structures are an advantage by themselves, since they are guaranteed to be bug-free[1]. But being able to formalize code that uses those data structures is also very important. Having precise semantics for our ADTs that are followed exactly by their implementation is the first step to achieve that, and it is the one we have focused on in this work. We, however, would like to show that the ADTs presented in this work are usable to solve real problems.

We have chosen two problems [2]: the first one is given to students learning how to use data structures and the second one is found in a repository of competitive programming problems. We have written a main method for the second one so that it can be compiled to an executable that can be executed with the rules set by competitive programming judges.

5.1 Reordering

Suppose that we are given a list of numbers ordered by their absolute value, for example, $[1, -1, -2, 3, -4, 5]$. We want to order them by their value (following the example, we would get $[-4, -2, -1, 1, 3, 5]$). The algorithm to achieve this is simple: we traverse the list separating negative numbers from positive numbers [3]. Then we first write the negative numbers in reverse order and then the positive numbers in the order we found them. The resulting list is ordered by value.

[1] Or, I should say, they follow the specification. There could still be bugs in the specification.

[2] Their solutions can be found in the `examples/` folder.

[3] Zero can go either to the negative numbers or the positive numbers, we choose one option.

```
method Reorder(neg: Stack, pos: Queue, v: array<int>)
  modifies v, neg, neg.Repr(), pos, pos.Repr()

  requires neg.Valid() ∧ neg.Empty() ∧ pos.Valid() ∧ pos.Empty()

  /* boilerplate preconditions and postconditions */

  requires ∀ i | 0 ≤ i < v.Length - 1 • abs(v[i]) ≤ abs(v[i+1])
  ensures Array.melems(v) = old(Array.melems(v))
  ensures ∀ i | 0 ≤ i < v.Length - 1 • v[i] ≤ v[i+1]
{
  Split(v, neg, pos);
  ghost var negmodel := neg.Model();
  ghost var posmodel := pos.Model();
  assert |negmodel| + |posmodel| = v.Length;  // proof omitted
  var i := 0;
  i := FillFromStack(v, i, neg);
  i := FillFromQueue(v, i, pos);
  LastLemma(negmodel, posmodel, v[..]);
}
```

Figure 5.1: `Reorder` method

To implement this algorithm we need two data structures, one for the negative numbers and one for the positive ones. The negative numbers will be extracted in reverse order, so the data structure that best fits the job is a stack. Positive numbers should be extracted in the same order we read them, so a queue will suffice.

Now that we know the general idea of the solution, we give (part of) the specification and the implementation of the `reorder` method in Figure 5.1. This method receives the auxiliary stack and queue that it uses internally so that we are abstracted away from the specific implementation of the traits `Queue` and `Stack` that we choose. The boilerplate code that we have omitted includes properties about the queue and the stack, such that they are different from each other, their representations are disjoint, etc. The specification makes sure that we return an ordered array that has exactly the same elements as before the call to the method. The implementation is as we sketched before: we separate negative and positive numbers and fill the array again first with the negative numbers and then with the positive ones. A final lemma is needed to prove that, assuming the models were both in increasing order, the resulting array will be in increasing order.

We will now continue with the methods that are called from `Reorder`. In Figure 5.2 we show the `Split` method. It pushes the negative numbers to the stack and enqueues the positive ones in the queue. Notice that the `Push` method

places the new element at the top of the stack, while the `Enqueue` method places the new element at the end of the queue. We will take advantage of this later, when we `Pop` and `Dequeue` from the front, having all the elements ordered the way we want. The implementation is a simple while loop that has to call the `TransitiveLemma` (shown in Figure 5.3) and add some annotations to prove that we are inserting the elements correctly.

Lastly, in Figure 5.4 we show the `FillFromStack` method. It receives the position i in the array where it will place the elements that it pops from the stack. The postconditions specify that the elements before i will stay the same, that the elements of the model will be placed in the array starting at position i, and that after those elements the array will stay the same. The method returns the position where it stops placing elements. This is necessary since `|st.Model()|` is ghost code (we can only know the length of the stack by popping all its elements). By knowing where the method left off, we can continue filling numbers from that position. The `FillFromQueue` method is very similar but with a queue.

5.2 Finding summits

The next problem we are going to explore is found in the *Acepta el reto* repository of competitive programming problems. The full problem statement can be found in Spanish in AER's webpage [4]. Here we will briefly summarize it. We are given a list of nonnegative numbers and our task is to find, for every position, its left adjacent summit. Two positions (a left summit and a right summit) are adjacent summits if (1) the element of the left summit is greater than that of the right summit and if (2) every element found between those two summits is smaller than both summits.

For example, suppose we are given the list $[3, 9, 2, 6, 5, 8, 7]$. Positions 1 and 3 (with elements 9 and 6, respectively) are adjacent summits, since $9 > 6$ and every element between them (only 2) is smaller than both. Notice that some positions do not have a left adjacent summit. Indeed, position 0 never has a left adjacent summit. We express that by returning -1 in that position. The complete list that we must return for this example is $[-1, -1, 1, 1, 3, 1, 5]$.

In Figure 5.5 we formalize the concept of adjacent summits and in Figure 5.6 we use it to specify the postcondition of the algorithm that solves the problem.

[4] Problema 571 found avaliable at `https://www.aceptaelreto.com/problem/statement.php?id=571`

```
method Split(v: array<int>, neg: Stack, pos: Queue)
  modifies pos, pos.Repr(), neg, neg.Repr()

  requires ∀ i | 0 ≤ i < v.Length - 1 • abs(v[i]) ≤ abs(v[i+1])

  /* boilerplate omitted */

  ensures ∀ x | x in neg.Model() • x < 0
  ensures ∀ i | 0 ≤ i < |neg.Model()| - 1 •
    abs(neg.Model()[i]) ≥ abs(neg.Model()[i+1])

  ensures ∀ x | x in pos.Model() • x ≥ 0
  ensures ∀ i | 0 ≤ i < |pos.Model()| - 1 •
    abs(pos.Model()[i]) ≤ abs(pos.Model()[i+1])

  ensures Seq.MElems(neg.Model()) + Seq.MElems(pos.Model()) = Seq.MElems(v[..])
{
  var i := 0;
  while i < v.Length
    /* invariants omitted (just the postconditions with slight variations) */
  {
    TransitiveLemma(v, i+1);
    assert ∀ j | 0 ≤ j < i • abs(v[j]) ≤ abs(v[i]);
    if v[i] < 0 {
      if |neg.Model()| > 0 {
        assert abs(v[i]) ≥ abs(neg.Model()[0]);
      }
      neg.Push(v[i]);
    } else {
      if |pos.Model()| > 0 {
        assert abs(pos.Model()[|pos.Model()|-1]) ≤ abs(v[i]);
      }
      pos.Enqueue(v[i]);
    }
    i := i + 1;
  }
}
```

Figure 5.2: `Split` method

```
lemma TransitiveLemma(v: array<int>, i: int)
  requires ∀ i | 0 ≤ i < v.Length - 1 • abs(v[i]) ≤ abs(v[i+1])
  requires 0 ≤ i ≤ v.Length
  ensures ∀ j, k | 0 ≤ j < k < i • abs(v[j]) ≤ abs(v[k])
{
  if i = 0 {
  } else if i = 1 {
  } else if i = 2 {
  } else {
    TransitiveLemma(v, i-1);
    assert abs(v[i-2]) ≤ abs(v[i-1]);
  }
}
```

Figure 5.3: Transitivity lemma

```
method FillFromStack(r: array<int>, i: nat, st: Stack) returns (l: nat)
  modifies r, st, st.Repr()
  requires st.Valid()
  requires i + |st.Model()| ≤ r.Length
  ensures r[..i] = old(r[..i])
  ensures r[i..i+old(|st.Model()|)] = old(st.Model())
  ensures r[i+old(|st.Model()|)..] = old(r[i+|st.Model()|..])
  ensures l = i + old(|st.Model()|)
{
  l := 0;
  while ¬st.Empty()
    invariant l = old(|st.Model()|) - |st.Model()|
    invariant st.Model() = old(st.Model()[l..])
    invariant r[..i] = old(r[..i])
    invariant r[i..i+l] = old(st.Model()[..l])
    invariant r[i+old(|st.Model()|)..] = old(r[i+|st.Model()|..])
  {
    r[i+l] := st.Pop();
    l := l + 1;
  }
  l := l + i;
}
```

Figure 5.4: `FillFromStack` method

```
predicate AdjacentSummits(v: seq<nat>, i: nat, j: nat)
  requires 0 ≤ i < j < |v|
{
  ∧ v[i] > v[j]
  ∧ ∀ k | i < k < j • v[i] > v[k] ∧ v[j] ≥ v[k]
}
```

Figure 5.5: `AdjacentSummits` definition

```
method FindSummits(v: array<nat>, st: Stack) returns (r: array<int>)
  modifies st, st.Repr()
  ensures v.Length = r.Length
  ensures ∀ i | 0 ≤ i < r.Length • -1 ≤ r[i] < i
  ensures ∀ i | 0 ≤ i < r.Length ∧ r[i] ≠ -1 • AdjacentSummits(v[..], r[i], i)
  ensures ∀ i | 0 ≤ i < r.Length ∧ r[i] = -1 •
    ∀ j | 0 ≤ j < i • ¬AdjacentSummits(v[..], j, i)
```

Figure 5.6: `FindSummits` method specification

The implementation in Figure 5.7 traverses the input list and fills the output array with the help of an auxiliary stack. We will begin by understading the invariant of the stack and will continue with the implementation properly. The stack contains all the positions that could be the left adjacent summit of a position that has not been traversed yet. To be precise, it contains a chain of adjacent summits starting at the position with the maximum number and ending at the position right before `i`. Notice that, by being adjacent summits, they do not have any bigger elements between them and they are in decreasing order. By virtue of that, if the element placed at position `i` is smaller than the top of the stack, the top of the stack will be the left adjacent summit of that position. If the element at `i` is bigger than the top of the stack, we can discard that element since it will not be the left adjacent summit of any position to the right. In fact we can remove every element in the stack that is smaller than `v[i]`. That task is given to the `RemoveLess` method, not shown here because it is a simple method that is mainly composed of boilerplate.

Now we can continue by describing the implementation. We will begin by pushing position 0 to the stack (if the input list is empty we just return). Inside the loop, after we remove every position in the stack whose element is smaller than the cursor by calling `RemoveLess`, there are two possibilities: if the stack is empty it means the cursor does not have a left adjacent summit; otherwise, on the top of the stack ther is the left adjacent of the top of stack and we mark it as such. Afterwards we push `v[i]` to the stack, since it could be the left adjacent summit of some other position later, and repeat until we reach the end of the list.

```
r := new int[v.Length];
var i := 0;
if r.Length > 0 {
  st.Push(0);
  r[0] := -1;
  i := 1;
} else {
  return;
}
while i < v.Length
  invariant 0 ≤ i ≤ v.Length
  invariant st.Valid()

  invariant ¬st.Empty()
  invariant st.Top() = i-1
  invariant ∀ j | 0 ≤ j < |st.Model()| • 0 ≤ st.Model()[j] < i
  invariant ∀ j | 0 ≤ j < |st.Model()|-1 •
    st.Model()[j+1] < st.Model()[j]
    ∧ AdjacentSummits(v[..], st.Model()[j+1], st.Model()[j])
    ∧ v[st.Model()[j+1]] > v[st.Model()[j]]
  invariant ∀ x | x in v[..i] • v[st.Model()[|st.Model()|-1]] ≥ x
  invariant ∀ j | 0 ≤ j < i • -1 ≤ r[j] < j
  invariant ∀ j | 0 ≤ j < i • r[j] ≠ -1 ⟹ AdjacentSummits(v[..], r[j], j)
  invariant ∀ j | 0 ≤ j < i • r[j] = -1 ⟹
    ∀ k | 0 ≤ k < j • ¬AdjacentSummits(v[..], k, j)
{
  ghost var max := v[st.Model()[|st.Model()|-1]];
  ghost var k := RemoveLess(st, v, i);
  assert ∀ x | x in v[..i] • max ≥ x;
  if st.Empty() {
    r[i] := -1;
    assert v[i] ≥ max;
    assert ∀ x | x in v[..i] • v[i] ≥ x;
    assert ∀ j | 0 ≤ j < i • v[j] in v[..i] ∧ v[i] ≥ v[j];
    assert ∀ j | 0 ≤ j < i • ¬AdjacentSummits(v[..], j, i);
  } else {
    r[i] := st.Top();
    assert st.Top() < i;
    assert v[st.Top()] > v[i];
    assert ∀ j | st.Top() < j < i • v[i] ≥ v[j];
    assert AdjacentSummits(v[..], st.Top(), i);
  }
  st.Push(i);
  i := i + 1;
}
```

Figure 5.7: `FindSummits` method implementation

Chapter 6

Iterators

Iterators are an important part of library software that had not been fully formalized yet in Dafny. They allow us to traverse an ADT without knowing its internal representation and without modifying it. They also allow us to modify certain elements or insert new elements in positions that are not directly accesible otherwise. Being able to formally verify their use and implementation is crucial if we want to verify whole systems of imperative programs.

In this work we have verified iterators in the `LinkedList` interface. This is not the general `List` ADT, but rather the public interface of a doubly linked list. In the `List` ADT we do not know when iterators are invalidated if the list is modified. In the `LinkedList` ADT, on the other hand, we specify exactly what is the state of the iterators after each method call. We have also verified iterators in the `ArrayList` ADT using arrays, but we will focus in this document on the `LinkedList` ADT.

6.1 Verification of linked list iterators

The verification of linked list iterators starts with the specification of the `Iterator` trait in Figure 6.1. Iterators are an ADT that have as model the `Index` at which they are pointing and the `Parent` list they are traversing. Iterators share their representation with their parent and have a representation invariant (`Valid`) that informs us if the iterator is usable or if it can no longer be used.

In terms of methods, two are notable: `HasNext` and `Next`. `HasNext` is the precondition of `Next`. It returns whether the index is less than the length of the model. Notice that this can trivially be expressed in the specification section of

```
trait ListIterator {
  function Parent(): List
    reads this

  predicate Valid()
    reads this, Parent(), Parent().Repr()

  function Index(): nat
    reads this, Parent(), Parent().Repr()
    requires Valid()
    requires Parent().Valid()
    ensures Index() ≤ |Parent().Model()|

  function method HasNext(): bool
    reads this, Parent(), Parent().Repr()
    requires Valid()
    requires Parent().Valid()
    ensures HasNext() ⟺ Index() < |Parent().Model()|

  method Next() returns (x: int)
    modifies this
    requires Valid()
    requires Parent().Valid()
    requires HasNext()
    requires allocated(Parent())
    requires ∀ it | it in Parent().Repr() • allocated(it)
    ensures Parent().Valid()
    ensures Valid()
    ensures old(Index()) < Index()
    ensures old(Parent()) = Parent()
    ensures old(Parent().Model()) = Parent().Model()
    ensures Parent().Iterators() = old(Parent().Iterators())
    ensures x = Parent().Model()[old(Index())]
    ensures Index() = 1 + old(Index())
    ensures ∀ it | it in Parent().Iterators() ∧ old(it.Valid()) •
      it.Valid() ∧ (it ≠ this ⟹ it.Index() = old(it.Index()))
    ensures ∀ x | x in Parent().Repr() - old(Parent().Repr()) • fresh(x)
    ensures ∀ x | x in Parent().Repr() • allocated(x)
}
```

Figure 6.1: Specification of the iterator trait

Dafny, but we cannot express it directly in the implementation section (writing `Index() < |Parent().Model()|`) since the model is not present at runtime. We need this `function method` to do a computation on real data. Once we know that the iterator is not pointing past the model, we can call the `Next` method that will return the current element of the iterator and advance its index by one.

Now we continue with the specification of the `LinkedList` ADT in Figure 6.2. The `Iterators()` set determines the iterators that the list has created. Notice that this set never decreases. When iterators get invalidated they are not removed from this set; their representation invariant just turns false. To create new iterators we call the `Begin` method. In the postcondition we need to specify that the old iterators remain the same. Similar postconditions will be present in all mutator methods.

With all those definitions we can see an example using iterators in Figure 6.3. It is a simple method that copies all the elements in a list into an input array. With the other ADTs we could not implement this method without emptying the list first (and then filling it again if desired). Thanks to iterators we can traverse the list without modifying it. The implementation is simple and should not need an explanation. The postcondition just specifies that the the model of the list is the same as the array with some boilerplate. The invariant is similar to the postcondition but it has to take the iterator into account. First, it must ensure that the iterator stays valid during the execution of the while loop. It must remain different to the other objects present; its parent is the list and its index is equal to the auxiliary variable. Then, as is usual in invariants, we include a partial postcondition: the model and the array until the i-th element are equal.

6.2 Invalidation of linked list iterators

Usually, in software libraries like those of Java and C#, iterators are invalidated after every mutation to the instance they are traversing. Using an iterator after such mutation could lead to exceptions or other undesired states. In the C++ Standard Library, on the contrary, the behaviour of iterators is fully specified in the case of linked lists and does not necessarily lead to exceptions: if the node an iterator is pointing to is removed, that iterator is invalidated (and it will probably raise some kind of exception, or fail silently); if it is not, that iterator can continue to be used. This kind of fine-grained specification of iterators is very useful when programmers want to write very performant code, but it is also a

```
trait List {
  function ReprDepth(): nat
  function ReprFamily(n: nat): set<object>
  function Repr(): set<object>
    reads this, ReprFamily(ReprDepth()-1)

  predicate Valid()
    reads this, Repr()

  function Model(): seq<int>
    reads this, Repr()
    requires Valid()

  function Iterators(): set<ListIterator>
    reads this, Repr()
    requires Valid()
    ensures ∀ it | it in Iterators() • it in Repr() ∧ it.Parent() = this

  method Begin() returns (it: ListIterator)
    modifies this, Repr()
    requires Valid()
    requires ∀ it | it in Iterators() • allocated(it)
    ensures Valid()
    ensures Model() = old(Model())
    ensures fresh(it)
    ensures Iterators() = {it} + old(Iterators())
    ensures it.Valid()
    ensures it.Index() = 0
    ensures it.Parent() = this
    ensures ∀ it | it in old(Iterators()) ∧ old(it.Valid()) •
      it.Valid() ∧ it.Index() = old(it.Index())
    ensures ∀ x | x in Repr() - old(Repr()) • fresh(x)
    ensures ∀ x | x in Repr() • allocated(x)
}
```

Figure 6.2: Specification of the `LinkedList` class

```
method FillArray(l: List, v: array<int>)
  modifies l, l.Repr(), v
  requires {v} !! {l} + l.Repr()
  requires l.Valid()
  requires v.Length = |l.Model()|
  requires ∀ x | x in l.Repr() • allocated(x)
  ensures l.Valid()
  ensures v[..] = l.Model() = old(l.Model())
  ensures ∀ x | x in l.Repr() - old(l.Repr()) • fresh(x)
  ensures ∀ x | x in l.Repr() • allocated(x)
{
  var it := l.Begin();
  var i := 0;
  while it.HasNext()
    decreases |l.Model()| - it.Index()
    invariant l.Valid()
    invariant l.Model() = old(l.Model())
    invariant it.Parent() = l
    invariant it.Valid()
    invariant {it} !! {l}
    invariant {v} !! {l} + l.Repr()
    invariant {v} !! {it}
    invariant it.Index() = i ≤ |l.Model()|
    invariant v[..i] = l.Model()[..i]
    invariant ∀ x | x in l.Repr() - old(l.Repr()) • fresh(x)
    invariant ∀ x | x in l.Repr() • allocated(x)
  {
    var x := it.Next();
    v[i] := x;
    i := i + 1;
  }
}
```

Figure 6.3: `FillArray` method with iterators

```
method PopBack() returns (x: int)
  modifies this, Repr()
  requires Valid()
  requires Model() ≠ []
  requires ∀ x | x in Repr() • allocated(x)
  ensures Valid()
  ensures Model() + [x] = old(Model())

  ensures ∀ x | x in Repr() - old(Repr()) • fresh(x)
  ensures ∀ x | x in Repr() • allocated(x)

  ensures Iterators() = old(Iterators())
  ensures ∀ it | it in Iterators() ∧ old(it.Valid()) •
    if old(it.Index()) = old(|Model()| - 1) then
      ¬it.Valid()
    else if old(it.Index()) = old(|Model()|) then
      it.Valid() ∧ it.Index() + 1 = old(it.Index())
    else
      it.Valid() ∧ it.Index() = old(it.Index())
```

Figure 6.4: `PopBack` method in the `LinkedList` trait

source of difficult to trace bugs. In this section we show the formal specification of iterators on linked lists and how users can make use of it to verify that their code never uses invalid iterators, allowing for fast and safe code.

As an illustrative example, let us see the specification of the `PopBack` method in Figure 6.4. It is what we would expect except for the last two postconditions. First, it is specifying that no iterators will be added (nor removed). Second, it specifies the behaviour of the existing iterators: if an iterator was pointing to the last node, it will get invalidated; if it was pointing past the end, it remains valid and its index is decreased by one; in any other case, it will remain valid and its index will remain the same.

Figure 6.5 shows an example where we reason about how iterators get invalidated or remain valid, even when their underlying collection is mutated. We pop the first element of the list (the head) with `PopFront` and `it1` gets invalidated because it pointed to the head, while `it2` remains valid.

6.3 Using iterators for element insertion

Lastly, we want to discuss iterators when they are used to insert elements. Figure 6.6 shows the interface for the `Insert` method in class `LinkedList`. It takes an iterator and uses it to insert an element in the middle of a list in constant time. In this work we have been able to formalize its implementation (as shown

```
var it1 := l.Begin();
var it2 := l.Begin();
var x := it2.Next();
var y := l.PopFront();
assert x = y;
assert ¬it1.Valid();
assert it2.Valid();
```

Figure 6.5: Example of iterator invalidation

in Chapter 3) and its use (as shown in this section).

The `Insert` method receives an iterator of type `ListIterator`, a trait. This is a problem since the implementation needs the underlying iterator class, not the abstract type of the trait. We could solve this with downcasting, since we can prove that `Insert` will only receive iterators of the concrete type of iterator that the implementation uses. However, as explained in Secton 2.4 this feature is not yet implemented. This part of the library will be incomplete for now because of that.

In Figure 6.7 we show an example of using this list to duplicate all the elements of a list. We traverse the list with an iterator and make use of the `Insert` method to insert a new element in-place without the need to pop any elements. An auxiliary lemma is needed to verify it, and some proofs have been omitted for presentation purposes. Notice that we use an alternative definition of the `DupRev` function that better matches the behaviour of the loop.

Why would we implement the `DupElements` this way? We could use the other mutator methods without sacrifing algorithmic complexity, but popping and pushing elements has a large memory footprint. With the `Insert` method, on the other hand, we only allocate the nodes that are strictly needed. Being able to verify efficient software as is written in industry is an important step to making formal verification mainstream.

```
method Insert(mid: ListIterator, x: int)
  modifies this, Repr()
  requires Valid()
  requires mid.Valid()
  requires mid.Parent() = this
  requires mid.HasNext()
  requires mid in Iterators()
  requires ∀ x | x in Repr() • allocated(x)
  ensures Valid()
  ensures Model() = Seq.Insert(x, old(Model()), old(mid.Index())+1)
  ensures Iterators() = old(Iterators())
  ensures ∀ it | it in Iterators() ∧ old(it.Valid()) • it.Valid()
  ensures ∀ it | it in Iterators() ∧ old(it.Valid()) •
    if old(it.Index()) ≤ old(mid.Index()) then
      it.Index() = old(it.Index())
    else
      it.Index() = old(it.Index()) + 1
  ensures ∀ x | x in Repr() - old(Repr()) • fresh(x)
  ensures ∀ x | x in Repr() • allocated(x)
```

Figure 6.6: `Insert` method interface

```
function Dup<A>(xs: seq<A>): seq<A>
{
  if xs = [] then []
  else [xs[0]] + [xs[0]] + Dup(xs[1..])
}

function DupRev<A>(xs: seq<A>): seq<A>
  ensures 2*|xs| = |DupRev(xs)|
{
  if xs = [] then []
  else DupRev(xs[..|xs|-1]) + [xs[|xs|-1]] + [xs[|xs|-1]]
}

lemma DupDupRev<A>(xs: seq<A>)
  ensures Dup(xs) = DupRev(xs)
{ /* ... */ }

method DupElements(l: List)
  modifies l, l.Repr()
  requires l.Valid()
  requires ∀ x | x in l.Repr() • allocated(x)
  ensures l.Valid()
  ensures l.Model() = old(Dup(l.Model()))
  ensures ∀ x | x in l.Repr() - old(l.Repr()) • fresh(x)
  ensures ∀ x | x in l.Repr() • allocated(x)
{
  var it := l.Begin();
  ghost var i := 0;
  while it.HasNext()
    decreases |l.Model()| - it.Index()
    invariant l.Valid()
    invariant 2*i ≤ |l.Model()|
    invariant i ≤ old(|l.Model()|)
    invariant l.Model()[..2*i] = old(DupRev(l.Model()[..i]))
    invariant l.Model()[2*i..] = old(l.Model()[i..])
    invariant it.Parent() = l
    invariant it.Valid()
    invariant {it} ↗↖ {l}
    invariant it.Index() = 2*i
    invariant it in l.Iterators()
    invariant ∀ x | x in l.Repr() - old(l.Repr()) • fresh(x)
    invariant ∀ x | x in l.Repr() • allocated(x)
  {
    assert i < old(|l.Model()|);
    ghost var omodel := l.Model();
    assert omodel[..2*i] = old(DupRev(l.Model()[..i]));
    var x := it.Peek();
    l.Insert(it, x);
    ghost var model := l.Model();
    assert model = old(Seq.DupRev(l.Model()[..i+1])) + omodel[2*i+1..];
// proof omitted
    x := it.Next();
    x := it.Next();
    i := i + 1;
  }
  assert l.Model() = old(Seq.Dup(l.Model()));  // proof using DupDupRev omitted
}
```

Figure 6.7: Duplicate the elements of a list with iterators

Chapter 7

Conclusions

In this chapter we will review the goals we set in Chapter 1 and evaluate the work that has been done. Regarding our goals, we have achieved them:

- We have developed a library of heap-based linked data structures in the form of ADTs that include stacks, queues, deques and lists.
- Such library follows a methodology that resulted from the study of different options for verifying ADTs, like the calculated model and structured or unstructured representation.
- Our methodology allowed for great code reuse: all of the ADTs could be verified with a handful of underlying data structures. It also maintained the abstraction of ADTs: the examples did not depend on implementation details and different implementations were developed for the same ADTs. Code reuse and abstraction were provided by the layered approach for ADT verification we have devised.
- The ADTs we have formalized can be used to solve real problems ergonomically, like the two competitive programming problems we verified in Chapter 5.
- We began exploration on the verification of iterators, with promising results for future work. Iterators are formally specified in such a way that allows fine-grained reasoning on what iterators have been invalidated. We also have in-place insertion of elements in lists thanks to iterators.

7.1 What I learned from this project

During this work I learned how to use Dafny to verify algorithms and data structures. I also learned the state of software verification using semiautomated theorem proving. It helped me see a different perspective to what I am used to (proof assistants and type theory), verifying imperative programs instead of functional ones, using automated methods instead of manual proofs.

This work was the first time that I had to write such a long document. It has advanced my writing capabilities greatly.

Above all, what had the most impact on me was that I began learning the process of academic research by establishing a research idea, studying the related literature, developing the idea and writing about it to publish it as an article. All of this taught me how to follow a more principled approach in my research interests.

7.2 Difficulties

It is known that formal verification is a harsh challenge. This work was not different. Dafny, even if it solves the majority of proof obligations with little help, can sometimes get stuck on simple properties and the programmer must spend hours trying to understand what Dafny needs to verify them. The conventions developed in Section 4.1 are the result of long scrutiny and communication with the main developers of Dafny.

Apart from the inherent problems of software verification, we also suffered the incompleteness of Dafny's reference manual on certain topics. The bugs and bad performance of Dafny did not help, neither. We could say that we pushed Dafny to its limits.

During this work a new release of Dafny was published. Trying to support different versions of Dafny at the same time was also a challenge.

7.3 Future work

We could advance this project in two directions:

- By applying the methodology devised in this work to verify other kinds of linked data structures like trees, maps, graphs, etc.

- By finishing work on iterators, implementing more methods that demonstrate their use and using the new features of Dafny that we need. We would also like to explore different approaches to iterator verification.

Bibliography

[1] The Agda Wiki. `https://wiki.portal.chalmers.se/agda/pmwiki.php`.

[2] Jorge Blázquez, Manuel Montenegro, and Clara Segura. Verification of mutable data structures in Dafny: methodological aspects. In *XX Jornadas sobre Programación y Lenguajes, PROLE 2021*, pages 1–16, 2021. To appear.

[3] Edmund M. Clarke. Model checking. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[4] The Coq Proof Assistant. `https://coq.inria.fr/`.

[5] Dafny Reference Manual. `https://dafny-lang.github.io/dafny/DafnyRef/DafnyRef`.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy. Correct code containing containers. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs*, pages 102–118, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] J. Harrison. Formal verification at Intel. In *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pages 45–54, 2003.

[9] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3), 2012.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[11] Isabelle's webpage. `https://isabelle.in.tum.de/`.

[12] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[13] K. Rustan M. Leino. This is Boogie 2. Technical Report KRML 178, Microsoft Research, http:// research.microsoft.com/ leino/papers.html, 2008.

[14] K.R.M. Leino. Developing verified programs with Dafny. In *VSTTE 2012*, pages 82–82, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[15] Rustan Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School 2008*, pages 1–36.

[16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016. SEE.

[17] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[18] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. *Formal Aspects of Computing*, 30(5):495 – 523, 2018.

[19] Rubén Rafael Rubio Cuéllar. Verificación de algoritmos y estructuras de datos en Dafny. TFG en Ingeniería Informática y Matemáticas (Fac. de Informática, UCM), `https://eprints.ucm.es/id/eprint/38702/`, 2016.

[20] Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design.* Morgan Kaufmann, 2015.